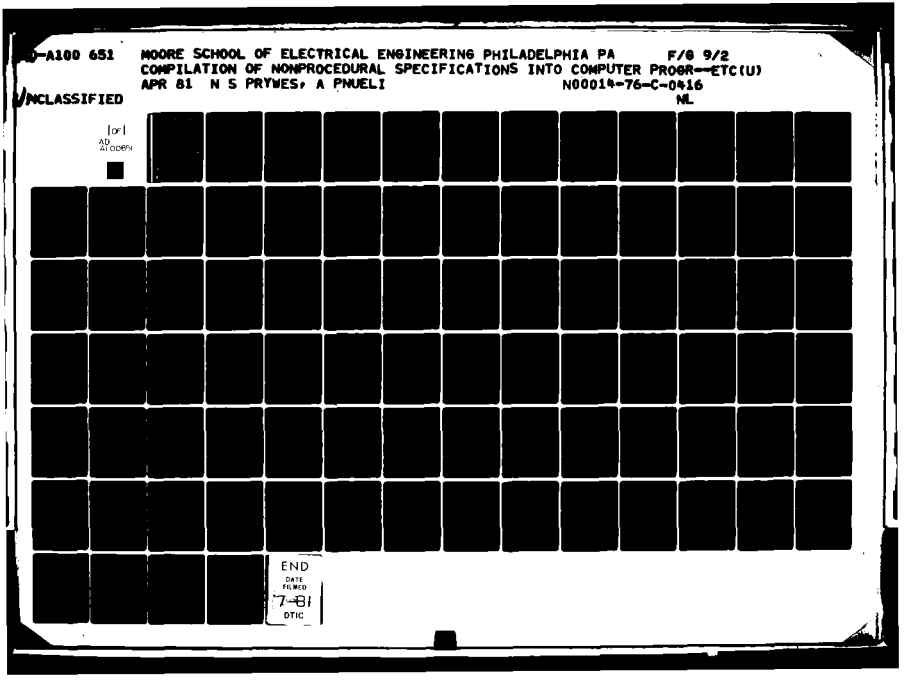

D-A100 651
MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
COMPILATION OF NONPROCEDURAL SPECIFICATIONS INTO COMPUTER PROGR--ETC(U)
APR 81 N S PRYWES, A PNUELI N00014-76-C-0416
UNCLASSIFIED NL
1 of 1
AD A100651
END
DATE FILMED
7-81
DTIC

LEVEL II

2

UNIVERSITY of PENNSYLVANIA

The Moore School of Electrical Engineering

PHILADELPHIA, PENNSYLVANIA 19104

UNIVERSITY of PENNSYLVANIA

PHILADELPHIA 19104

The Moore School of Electrical Engineering D2

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

9 Technical Report

COMPILATION OF NONPROCEDURAL SPECIFICATIONS INTO COMPUTER PROGRAMS.

by

N.S. Prywes and A. Pnueli

1 April 1981

Prepared For

Information Systems Program

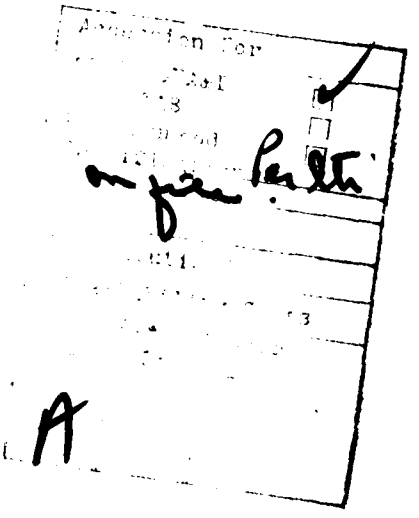
Office of Naval Research

Under Contract No. N00014-76-C-0416

15

Automatic Program Generation Project

TABLE OF CONTENTS



LIST OF FIGURES

LIST OF TABLES

COMPILATION OF NONPROCEDURAL SPECIFICATIONS INTO COMPUTER PROGRAMS*

N.S. Prywes
Department of Computer and Information Science, Moore School,
University of Pennsylvania, Philadelphia, Pa. 19104

A. Pnueli
Weizmann Institute, Rehovot, Israel

ABSTRACT

The article describes the compilation of a program specification written in the Very High Level nonprocedural MODEL language into an object (PL/I or Cobol) procedural language program. Nonprocedural programming languages are descriptive and devoid of procedural controls. They are therefore easier to use and require less programming skills than procedural languages. First, the MODEL language is briefly presented and illustrated. An important phase in the compilation process is the representation of the specification by a dependency graph denoted as array graph which expresses the interdependency between statements. Two classes of algorithms which utilize this graph representation continue the compilation process. The first class checks various completeness, non-ambiguity and consistency aspects of the specification. Upon de-

*Research supported by the Information Systems Program, Office of Naval Research, Contract No. N00014-76-C-0415.

tecting any faults the system attempts some automatic correcting measures which are reported to the user. Alternately when no feasible corrections are possible it reports an error and solicits a user modification. The second class of algorithms produces a general design of an object program in a language independent form. Finally PL/I or Cobol code is generated, based on the general design.

The algorithms are described informally. A number of less important algorithms are omitted, including the algorithms used to generate PL/I or Cobol code based on the intermediate design. A complete documentation of the system is available in the references.

Index terms: Nonprocedural languages, Very High Level Languages, Program Specifications, Compilers and Generators, Automatic Program Generation, Data-flow Languages.

Computing Reviews Categories: 4.12 and 4.22.

I. INTRODUCTION

This paper describes the process of compiling a program specification, written in the very high level nonprocedural MODEL language, into a program in a conventional high level procedural language, such as PL/I or Cobol.

Nonprocedural languages have been proposed for over a decade as more natural, more reliable and easier to use than procedural languages (Tessler and Enea, 1968; Leavenworth and Sammet, 1972; Ashcroft and Wadge, 1977). The advantages in use of the MODEL nonprocedural language and its processor have been discussed in a previous paper (Prywes, Pnueli, and Shastry, 1979). Some of the considerations reported there are repeated here very briefly. A nonprocedural language has no need for the procedural and control constructs of a conventional procedural language and the order of presentation of the language statements has no significance. The user of a nonprocedural language concentrates only on describing data, independently of the medium of the data (i.e. memory, data base or any other external storage), and on composing equations that define output variables in terms of input variables. Consequently, the user concentrates on expressing his program in a way which is most natural for the given problem, and is not distracted by the need to design

efficient representations and algorithms. Within a reasonable framework the task of design for efficiency is automatically undertaken by the system. A nonprocedural specification is therefore much shorter than the equivalent procedural program. The computer proficiency required of the user is also reduced through the elimination of the procedural design and considerations of efficiency. The aggregate of descriptive statements is unlike a procedural program; therefore it is inappropriate to refer to is as a "program." Instead, we use the word specification. The programs produced by the MODEL processor are more reliable. The task of debugging is carried out on a much higher level, verifying only the correctness of the specifications, and hence is much simpler than debugging a procedural program.

Previous developments of processors for nonprocedural languages have taken the interpretation route. While this approach can ensure flexibility and generality of the nonprocedural language, the resulting system usually suffers a decrease in efficiency when executed on a conventional machine. Also, the diagnostic capability of an interpreter is usually poorer in that very little preliminary analysis is attempted. Therefore, in our development of the MODEL nonprocedural language and processor, we have taken the compilation route, translating a specification into a conventional high level language. Also we have incorporated the capability for handling data bases and in-

put/output required for realistic applications. Due to the nature of the nonprocedural language, and the need to schedule program events, the process of compilation in this case is unlike that of conventional procedural languages. Several of the more important problem areas in the design of the MODEL processor are briefly described below.

A specification in the MODEL language consists of two types of statements: data description statements which describe the structure and attributes of variables, and equations defining some variables (the dependent variables of the equations) in terms of other variables (the independent variables of the equations). Some of the variables are designated as source variables and some as target variables. The role of the specification is to describe the transformation between source and target values. Typically, both source and target variables are located in external files.

Consider now the basic problems of translating such a specification into a conventional program:

Unordered nature of the specification: The order of the statements in the specification is not significant. Consequently, it is necessary to analyze the specification globally to determine all the dependencies between variables and equations which imply a partial ordering of the events in the program. Thus, an equation for defining a

variable can be calculated only when all the independent variables, i.e. variables appearing on the right hand side of an equation, are already defined. Consequently the calculation of an equation should be preceded by the calculations of all the equations defining the independent variables of this equation. Similarly, the instructions for reading the value of a variable which resides on secondary storage should precede any equation for which this variable is an independent variable. The array graph representation of the specification shows this precedence order between statements. This graph is also analyzed in order to detect circular dependencies, and then used to synthesize the program by translating statements in an order consistent with the precedence constraints.

Handling input/output: The user description of data is independent of the medium of the data and whether its representation is internal (in core) or external (secondary storage). It is necessary then to determine, based on file descriptions or the dependencies between variables and equations, whether the data is on an input/output device or in main memory, and if necessary, schedule the associated input/output instructions.

Analysis of repetitive equations and loop design: Since the language allows structured variables which may be tree-structured or arrays, an equation for such a variable

defines an aggregate of values. In the case of an equation defining an array variable, The translation calls for repetitive calculation of the equation for different values of the subscripts which explicitly or implicitly subscript the equation. This implies that the translation will enclose the equation within repetitive loops, which might be nested if the array is of multiple dimensions. In constructing the loops we must perform a deeper analysis of the interdependency between variables. In the presence of array variables we may have elements of one array dependent on elements of another array in a complicated manner, as well as the possibility that one element of an array may depend on another element of the same array. These considerations require that the elements of the arrays be computed in a certain order, and hence impose constraints on the loop design.

Checks and diagnostics: In contrast to the situation with procedural programming languages, most errors in a nonprocedural language stem not from coding errors but from mathematical incompletenesses or inconsistencies. Detected errors must be communicated to the user in strictly nonprocedural terms (i.e. without referencing program design considerations). The compilation process incorporates methods which resolve the problems

automatically and report the corrections to the user, as an aid in explaining the respective detected problems. This methodology of program checking and communication with the user goes beyond today's compilers of procedural languages.

Efficient use of memory: As discussed earlier, in specifying the data description statements, the user chooses the description which is most natural and appropriate for the problem. This choice does not necessarily lead to the most efficient data representation. It is up to the processor to map the conceptual structure onto a physical memory layout. In this mapping it is necessary to analyze the possibility of sharing of storage by different structures or even by different parts of the same structures. This is particularly important for large external data bases, where it is frequently mandatory to bring into memory at most one or a few records at a time.

The discussion in this paper follows the flow of control in the MODEL system. The input to the system is a program specification in the MODEL language. The syntax and semantics are briefly described in Section 2 together with two examples, which are used through the paper to illustrate the compilation process. The

language processor has five major phases:

1) Syntax analysis

2) Representation of the specification and the dependencies between its elements by an array graph. This is described in Section 3. An array graph is a compact representation of a large structured graph and is used here to represent the dependencies. The basic algorithms of Graph Theory can, under appropriate restrictions, be carried out on the array graph as well.

3) Consistency checking and correction of the specification. This is described in Section 4. The algorithms in this phase detect missing definitions, resolve ambiguities in naming of variables and verify consistency of dimensionality, range and subscripting. Many of the interactions with the user, utilizing nonprocedural terms, occur in this phase.

4) Generation of a flow chart for the program. The general design of a program is described in Section 5. These algorithms sequence the instructions implied by the data structures and equations. Iterations are designed to reduce memory and time costs. Program optimization is based on the notion of maximizing the scope of the iterations, particularly those that incorporate input or output operations.

5) The generation of PL/I or Cobol code is based on

the design generated in phase 4.

The first and last phases, which we consider less novel, have been omitted from our discussion. The present paper is based on an operational version of the MODEL system which is described in detail in a reference (MODEL Program Generation: System and Programming Documentation, 1980). Ongoing research on several improvements is described in Section 6.

2. THE MODEL SPECIFICATION LANGUAGE

A specification in the MODEL language consists of an unordered set of statements. The statements in the language are of two types: data description statements and equations which we call assertions. The data description statements describe the structure and attributes of the variables participating in the specification. The assertions define the values of some variables in term of other variables. The variables appearing in a specification are designated as source variables or target variables in header statements. The header statements are not important to the discussion here and are omitted in the following[1]. The values of the source variables are considered to be available on external input files. Target variables are to be produced on external output or update files. Target variables may alternately be designated as interim, to indicate that they need not be retained as output. The two subsections below describe the syntax of data and assertion statements respectively. Two examples are used to illustrate the composition of these two types of statements.

2.1 Data Statements

Data in a MODEL specification may be highly structured. The description of the data structure is tree-

1 Several features that provide additional ease have been omitted. For a more complete description of the language refer to MODEL II User Manual, 1978.

oriented, similar to PL/1 or Cobol. The node at the root of the data structure tree typically represents a file. A file may be composed of substructures, each of which may be further composed of substructures, and so on. A substructure is referred to as the parent of its component substructures. The latter are referred to as descendents. A data structure is visualized as a tree where substructures form nodes with branches leading to lower level components. The syntactic definition of data statements is shown in Figure 1. The < data name > is the name of a node in the tree. The < node type > indicates a level in the tree. A FILE node type may only appear at the root of the tree. A terminal tree node is denoted as FIELD node type. An intermediate node in the tree which is also the unit of transfer of data between input/output and memory is of RECORD node type, as in PL/I or Cobol. A GROUP node type is any other intermediate node in a tree.

The optional < file arguments > describe the computer media of the data[2]. They are unimportant to the dis-

2 File arguments are necessary for generating a Cobol Program. For a PL/1 program the medium may be specified in the JCL statements.

< data statement > : = < data name > IS < node type > (<arguments>)

< node type > : = FILE | GR[OU]P | REC[ORD] | F[IE]LD+

< arguments > : = < file arguments > | < group/record arguments > | < field arguments >

< group/record arguments > : = < immediate descendent name > [(< number of repetitions>)] [, < immediate descendent name > [(< number of repetitions >)]]*

The square brackets ([X]) denote optionality; when followed by an asterisk ([X]*) they mean zero or more repetitions.

+ The node type may be preceded by the key word INT[ERIM] when the respective data structure is target data but is not needed on an output medium.

Figure 1 Major Syntactic Components of Data Statement

cussion below and will be omitted in the following.

The number of repetitions of a descendant structure is included as an argument in the statement describing the parent. If the descendant occurs only once, then the < number of repetitions > is omitted. If the number of repetitions varies, then the minimum and maximum bounds may be specified. Also, unknown number of repetitions may be specified by an asterisk (*) in place of a repetition count[3]. The definition of a variable number of repetitions is further discussed below.

The field arguments are: data type, size and scale, with the same meanings these attributes have in PL/1. They are omitted in the following.

The example in Figure 2 illustrates a business application, which characteristically includes input/output. It consists of processing source sale documents to produce a monthly sales report. The data statements are in lines d1 to d14. Line d1 describes the IN sale source data. IN IS FILE (INGRP(*)) means that the file IN consists of an unspecified sequence of repetitions of struc-

3 In specifying an asterisk, the user implies to the system a memory allocation scheme in which only a few elements are retained in memory. This requires primarily limiting subscript expressions to the form I - K, for the respective dimension. This point is discussed further in Section 2.2. We are currently developing a new version which would perform this task automatically (see Section 6).

```
/*  DATA DESCRIPTION OF IN FILE */

d1:       IN IS FILE(INGRP(*))
d2:          INGRP IS GRP(INREC(*))
d3:             INREC IS REC(ITEM#,QUANT)
d4:                ITEM# IS FIELD
d5:                QUANT IS FIELD

/*  DATA DESCRIPTION OF ITEM FILE */

d6:       ITEM IS FILE(ITEMREC)
d7:          ITEMREC IS REC(ITEM#,PRICE)
d8:             ITEM# IS FIELD
d9:             PRICE IS FIELD

/*  DATA DESCRIPTION OF OUT FILE */


d10:      OUT IS FILE(OUTREC(*))
d11:         OUTREC IS REC(ITEM#,TOTAL,COST)
d12:            ITEM# IS FIELD
d13:            TOTAL IS FIELD
d14:            COST IS FIELD

/*  ASSERTIONS FOR DATA PARAMETERS */

a1:   IF END.INREC(FOR_EACH.INREC)
      THEN POINTER.ITEMREC = IN.ITEM#(FOR_EACH.INREC)
a2:   END.INREC = (IN.ITEM#¬=NEXT.IN.ITEM#)

/*  ASSERTIONS FOR OUT FILE DATA */

a3:   OUT.ITEM# = ITEM.ITEM#
a4:   TOTAL = SUM(QUANT(FOR_EACH.INREC), FOR_EACH.INREC)
a5:   COST = PRICE *TOTAL
```

Keywords are underlined.

Figure 2 MODEL Specification for Producing a Sales Report

tures named INGRP. In line d2, INGRP IS GRP (INREC(*)) means similarly that INGRP is a group consisting of an unspecified number of INREC structures. Line d3 shows that INREC is a RECORD containing information on quantity, QUANT, of the item sold, identified by ITEM#. The PRICE of each item is in another source file ITEM (lines d6 to d9). The target data is a summarized sales report named OUT (lines d10 to d14). Each record in OUT contains the ITEM#, TOTAL, and COST. TOTAL is the sum of all the quantities (QUANT) of an item of a specific valued ITEM#, that have been sold. COST is the product PRICE*TOTAL. This example is further explained in connection with later discussion of the assertions.

Although data are pictured in MODEL (as in PL/1 and Cobol) as tree structures, it will be more convenient for the discussion here to refer to data as <u>arrays</u>. There is a direct correspondence between the tree and array views of a data structure. For instance, specifying a <number of repetitions> means that the data structure repeats, constituting a vector. Generally, a structure may be viewed as a multidimensional array, where < number of repetitions > specifications of own or predecessor nodes in the data tree give the ranges of respective dimensions. Thus for instance, ITEM# and QUANT in the IN file are viewed as two dimensional arrays. The first, more

significant dimension corresponds to repetitions of INGRP and the second dimension corresponds to repetitions of INREC. Therefore, we refer in the following to the < number of repetitions > of a node as a range specification, and also as the range of the dimension. Viewing the data as arrays allows referring to a specific instance of the data as an element of an array which can be identified by the appropriate indices for each dimension. For instance ITEM#(n1,n2) denotes the ITEM# in the n2 th INREC of the n1 th INGRP. Element indices are denoted by free subscript variables that may assume integer values in the range of the respective dimension.

The range of a dimension may depend on the values of higher order subscripts. Therefore the range of a dimension of an array may not have the same value for all higher order dimension indices. Such an array is not rectangular and is referred to as a jagged edge array. For example, INREC has two dimensions with variable ranges associated with the repetitions of INGRP and INREC. The number of INREC instances varies from one instance of the parent INGRP to another. INREC may be viewed as a two dimensional jagged edge array, with a row corresponding to each instance of INGRP and the INREC instances corresponding to elements of the respective rows. Since the number of INREC instances varies from row to row (i.e. from one INGRP group

to another), the resulting matrix is not rectangular, but jagged edge.

Referring to an element through subscripting, and defining a variable range by use of an assertion are further discussed below in connection with the use of assertions.

The example in Figure 3 defines testing the primeness of an integer N, and if prime, the derivation of one divisor (DIV) of N.[4] The IN source file (lines d1 to d3) contains a single record with the variable N, and the OUT target file (lines d4 to d7)contains a single record with N and DIV.

The algorithm evaluates progressively the products of two integers for the purpose of testing equality to N. The product of the two integers is represented then by J (lines d8 to d10). Note that in line d10 of Figure 3, J is stated to be INTERIM, namely it is target data but the user is not interested in retaining J. It also means that J is needed for ease in specifying the algorithm for testing primeness but is not part of the desired result.

[4] It is similar to the testing of primeness example used in a description of the LUCID nonprocedural language (Ashcroft and Wadge, 1977). The choice of the same example should help the interested reader to compare LUCID and MODEL

```
                /* DESCRIPTION  OF  IN  FILE */

d1:     IN IS FILE(INREC)
d2:        INREC IS REC(N)
d3:              N IS FIELD

                /* DESCRIPTION OF OUT FILE */

d4:     OUT IS FILE(OUTREC)
d5:         OUTREC IS REC(N,DIV)
d6:           N IS FIELD
d7:              DIV IS FIELD

               /*  DESCRIPTION OF INTERIM DATA */

d8:     INT IS GRP(I(*))
d9:         I IS GRP(J(*))
d10:          J IS INTERIM FIELD

               /*  ASSERTIONS FOR DEFINING END.I AND END.J */

a1:     END.J = (J ≥ N)

a2:     IF END.J(SUB1 )THEN END.I =((J(SUB1) = N)∨(SUB1=1)))

              /*   ASSERTION FOR DEFINING J */

a3:     IF SUB1 > 1
           THEN J(SUB2, SUB1)=J(SUB2,SUB1-1)+SUB2+1
           ELSE J(SUB2, SUB1)=(SUB2+1)**2

              /*   ASSERTIONS FOR DEFINING VARIABLES IN OUT FILE */

a4:     IF END.I(SUB2) ^
                  (J(SUB2,SUB1)=N)
                THEN DIV = SUB2+1
                ELSE DIV = 'PRIME'

a5:     OUT.N = IN.N
```

Figure 3: MODEL Specification For Testing
The Primeness of An Integer

2.2 Assertion Statements

While the data statements describe the existence and structure of data to be operated upon, the description of the transformations applied to the data is given by the assertions. Rather than give detailed procedural instructions on step-by-step execution, the user of MODEL identifies relationships between the variables, from which the processor deduces the actual execution sequences. These relationships are called assertions in MODEL. The building blocks for assertions include conventional arithmetic and boolean expressions and more structured operations such as IF-THEN-ELSE. This section describes the syntax and semantics of assertions with the aid of the two examples in Figures 2 and 3. The focus is on the properties of special variables that define parameters of data, subscripts and functions.

The syntax used for assertions in this paper is the same as that of computation statements in PL/1. The language allows explicit equality relations of the form:

< variable > = < expression >

The variable on the left hand side, the dependent variable of the assertion, is defined by the expression on the right hand side. The independent variables for this assertion are the variables participating in the defining ex-

pression on the right hand side. An expression is built out of variables and constants to which are applied basic operators and functions. PL/1 conventions for constants, variables and boolean and arithmetic operators are used in composing expressions. These include the IF-THEN-ELSE operator whose syntax is:

IF < condition > THEN < variable > = <expression_1>
ELSE < variable > = <expression_2>,

meaning that if <condition> evaluates to TRUE, then < expression_1 > defines the value of the variable, otherwise < expression_2 > is used. An assertion defines only one variable and therefore the same variable name must be used following the THEN and ELSE keywords.[5]

An assertion statement, though similar in syntax to an assignment statement in procedural languages, should be regarded by the user quite differently. The assertion meaning is identical to the mathematical notion of equivalence between the two sides of the equal sign. Namely it is an equation. This aspect is basic to the difference between procedural and nonprocedural languages.

---

5 An alternative Algol-like syntax: < variable >= IF < condition > THEN < expression_1 > ELSE < expression_2 >, is also available. This syntax shows more clearly the equation quality of an assertion.

Because of the nonprocedural nature of MODEL, each variable name may denote only one value. Also the "historical" values of data, namely those that would not be needed further in a computation must be explicitly represented by symbolic names. In contrast, procedural programming languages allow assigning differing values to the same variable and "historical" values may be discarded if not further needed. For instance,an assignment statement within a loop: X=X+1 would make no sense as an equation. In MODEL it would be necessary to name each value of X separately. Assume that these values constitute a vector, with N elements. An element is denoted by subscripting: X(I). I is the subscript variable which can take the value of an integer in the range of 1 to N.[6] The MODEL equivalent of the above assignment statement is the assertion: X(I)=X(I-1)+1.

Both the dependent and the independent variables should be subscripted by a list of subscript expressions corresponding to the dimensions of the variables as specified in the data description. Any integer valued ex-

---

6 The more general case is where with each dimension we associate a lower limit ℓ_d, an upper limit u_d and an increment c_d. The node $X(I_1, I_2 \cdots I_m)$ then may have the form $(\ell_1,u_1,\ell_2,u_2,c_2 \ldots\ldots \ell_m,u_m,c_m)$. The more general case is handled by Shastry (1978).

pression can be used as a subscript expression for the variables. The general syntax for subscripted variables is:

< element of array > ::= < field name >

(<subscript expression>[,<subscript expression>]*)

The subscript expressions must be ordered according to the dimensions. Free subscript variables, as well as other variables and constants and arithmetic operations may be used in composing subscript expressions.

A free subscript variable may be global to an entire specification or local to an assertion. The same global subscript name in a number of assertions refers to free subscript variables of the same range. Global subscript names use the syntax form of FOR_EACH. <data name>. They may then have any integer value in the range of the <number of repetitions> associated with the <data name>. For instance assertion a5 in Figure 2 can be written using global subscripts as: COST(FOR_EACH.INGRP)=PRICE(FOR_EACH.INGRP)*TOTAL(FOR_EACH.INGRP). Use of the same local subscript name in different assertions does not imply referring to free subscript variables of the same range. Local subscript names use the syntax form of S[UB]<n>. Using local subscripts, assertion a5 of Figure 2 could be written as COST(S1)=PRICE(S1)*TOTAL(S1). Either representation would

be acceptable. The use of local subscripts is easier in many cases as the user need not consider the ranges of dimensions of different data structures. The syntax of a global subscript name is somewhat awkward and a shorter global subscript name, such as commonly used symbols for subscripts, I,J,K etc., may also be declared. The syntax for declaring a global subscript name is:

<subscript names>{IS / ARE}SUBSCRIPT (<number of repetitions>)

Subscript expressions are classified into four types according to use of the following syntactic forms:

1) <free subscript variable>
2) <free subscript variable>-1
3) <free subscript variable>-K, K is integer >1
4) Any form of arithmetic expression except types 1, 2 and 3 above.

The user is advised to give preference to use of subscript expressions of types 1, 2 and 3, as the version of the MODEL system reported here analyses the correctness of the specification and endeavors to obtain efficiency of the resulting program more thoroughly when these types of subscript expressions are used.

The subscripting of variables is a complex task that is difficult for many users. Subscripts may be implicit in cases which do not lead to ambiguity. Allowing omission of such subscripts eases the composition of assertions.

Following are the subscript usages that must be specified:

1) Subscripts used in subscript expressions of types 2, 3 and 4 (see above).

2) Subscripts of dimensions that are reduced or added in an assertion (i.e., where an independent variable has more or less dimensions than the dependent variables).

3) Once a subscript is specified in an assertion it must be consistently specified with all the variables in the assertion where the subscript applies.

4) Subscripts on the right of any specified subscripts.

5) Missing local subscripts are assumed inserted in all variables of an assertion monotonically (i.e., S1,S2...) from right to left. Subscripts must be specified in cases where this assumption is not valid.

Subject to these rules, the MODEL system performs analysis to insert missing subscripts. Thus assertion a5 in Figure 2 is stated as COST=PRICE*TOTAL, omitting the subscripts altogether. Figures 2 and 3 omit some subscripts (using global subscripts in Figure 2 and local subscripts in Figure 3). This will be further discussed below.

Of particular interest in the following are the use of qualified names and function names in assertions.

They are first briefly presented and thereafter further discussed with the aid of our two examples.

Qualified names may be used in assertions, using a period (.) to connect individual names (similar to PL/1). The most common use of a qualified name is to eliminate ambiguity through prefixing a name of a higher level structure. For instance in the example in Figure 2, there are three ITEM# variables, in files IN, ITEM, and OUT. They are unambiguously referred to in assertions a1 and a3 as IN.ITEM#, ITEM.ITEM# and OUT.ITEM# respectively.

Another common use of qualified names is to eliminate ambiguity in data that are updated. The keywords OLD and NEW are used then. For instance an assertion NEW.PRICE(J)= OLD.PRICE(J)+INCREMENT would update the PRICE in the ITEM file in Figure 2. An update of a file is visualized as creating a new version of the file, which would add a dimension to the file structure. This is difficult to use, and use of OLD and NEW keywords is preferred.

There are parameters of the data structures which depend on values of source or target variables. We refer to these as data parameter variables. Characteristically, these parameters provide specifications for sizes of arrays, lengths of character strings, keys for access to files, etc. They introduce to MODEL the flexibility of variable size or

dynamic structures. The syntax of a data parameter variable is:

<data parameter variable> ::= <reserved keywords>.<variable>

Data parameter variables may be explicitly defined by assertions. They may denote entire arrays and be used with subscript expressions in the same way as other variables. These keywords are listed below and further discussed in the sequel.

END.<data name>	denotes whether the named data element is the last one in the range of a dimension.
ENDFILE.<file name>	denotes an end-of-file marker of the named file.
FOUND.<record name>	denotes existence of the record in an index sequential file that is accessed through a POINTER variable (see POINTER below).
LENGTH.<field name>	denotes length of the named field.
NEXT.<field name>	denotes a named variable in the next adjacent record on the medium source data.
POINTER.<record name>	denotes value of a key used to reference a keyed record in an index sequential file. (The key name is identified in the FILE statement.)
SIZE.<data name>	denotes the range of the lowest order dimension of the repeating data structure named in the suffix.

These variables are [INT]ERIM , i.e., they are not output, but are otherwise considered same as target data. Data

description statements for these variables may be provided optionally. If not provided, each of these variables will be automatically assigned the appropriate dimensionality. These variables are further explained below.

When the range of a dimension is variable, the range is viewed as denoted by an auxiliary array variable which may be defined by an assertion. A variable range data structure X may have its range denoted by a structure named SIZE.X, of one dimension less than that of X (the rightmost) and same ranges of the other dimensions. Thus if X is m dimensional the elements of SIZE.X have the values of the ranges of the lowest order dimension of X for each of the higher order dimensions indices. Thus I_m, the subscript for the m-th dimension of $X(I_1....I_{m-1}, I_m)$ must be in the range $1 \leq I_m \leq SIZE.X(I_1...I_{m-1})$. consequently if the values of the elements of SIZE.X are not equal, then X is not a rectangular array but a <u>jagged edge</u> array. The range must be ≥ 0.

Another option for defining the size of structure X is by an auxiliary boolean array named END.X that has the same dimensions and ranges as X. A 0 value of an element of X denotes that it is not the last element within the range of the rightmost dimension, and a 1 denotes that it is the last element. When END.X is used for range speci-

fications then the range must be ≥ 1.

For example the ranges associated with INGRP and INREC in Figure 2 could be denoted by END.INGRP and END.INREC respectively, The termination of the INGRP structure in file IN can be determined by an end-of-file marker at the end of file IN. The definition of END.INGRP is therefore implicit and the user may omit defining this variable by an assertion. Alternately, ENDFILE.IN variable denotes recognition of end-of-file marker on the file medium, and it could have been used to define END.INGRP, but as noted above this definition has been omitted in Figure 2. a2 in Figure 2 defines END.INREC. This is further explained below.

POINTER.<record name >, defines an access key to an index sequential or random access file. The file ITEM described in lines d6-d9 of Figure 2 is an index sequential file[7]. POINTER.ITEMREC is a vector with an element for each instance of INGRP (the FOR_EACH.INGRP subscript is implicit and has been omitted in assertion a_1). Let us represent assertion a1 by: POINTER.ITEMREC(I)= EXPR(I). The array of records ITEMREC is considered as

7 The sorting order and file organization can be optionally provided by the user in the file arguments, which have been omitted in this paper.

indexed in the order of the elements of the retrieval keys POINTER,ITEMREC. Namely, the record retrieved by using EXPR(I) as a key is considered to be the I-th element in the array ITEMREC.

Finally, function references can be made to denote an operand in assertions. The built-in functions of PL/1 may be used with the MODEL program generator that produces PL/1 object programs. There is a subset of the PL/1 built-in functions in the version of the system that produces Cobol object programs. Additional functions may be coded in the object language and placed in the system function library.

Let us now consider in full the examples in Figures 2 and 3. The specification in Figure 2 describes a business application which processes source sale documents IN to produce a monthly sales report OUT. The user may designate IN as source data and OUT as target data in a separate header section of the specification. Discussion of a header section has been omitted in this paper. Alternately, lack of assertions defining the variables in IN would imply that IN is a source file, and the existence of defining assertions implies that OUT is a target file. Lines d1 to d5 describe the IN file as a two dimensional array. Assume in this specification that the sales records are sorted by ITEM#[7] so that all the records with the same ITEM# value

appear contiguously. Consequently we conveniently view the file as an array of groups INGRP, each such group being an array of records with identical ITEM# values. This grouping is conceptual rather than physical. We need an assertion which determines the range of INREC instances based on comparison of ITEM# values in consecutive records. Assertion a2 is responsible for this determination. END.INREC has the same dimensions and ranges as INREC. It denotes the last element of INREC. The last INREC record (of an INGRP group) is recognized by the change of the item number. The ITEM# in a subsequent record is referred to as NEXT.ITEM#.[8]

For each INGRP group we would like to sum all the sale quantities QUANT associated with a given item. This is done in a4. The SUM function sums elements along one dimension of an array. In this case the elements of QUANT are summed along the second dimension. Note that the subscript for the first dimension is implicit and has been omitted. The function SUM is referred to as a reduction function as the number of its dimensions is one less than the number of dimensions of its argument. We then

---

[8] Note that NEXT.ITEM# may be in the next group and have an element index 1. Thus, NEXT.ITEM# is not the same as ITEM# (FOR_EACH.INGRP, FOR_EACH.INREC+1).

calculate the total cost of sales for this item by multiplying TOTAL by PRICE. However, the PRICE information resides on an auxiliary index sequential file ITEM . The ITEMREC with the relevant PRICE is referenced defining the ITEM# field as a key. The fields in an OUT record are defined in assertions a3-a5.

As noted, the fully subscripted form of assertions requires writing down long subscript lists for almost every variable. In order to alleviate this chore somewhat we allow some subscripts to be omitted in Figure 2. This considerably simplifies the assertions. The assertions in lines a1 to a5, Figure 2, use global subscripts. The subscript FOR_EACH.INGRP can be omitted in all assertions.

In a1 POINTER.ITEMREC denotes the value of a key that associates an instance of ITEMREC with an instance of INGRP that has the same value of ITEM#. POINTER.ITEMREC as well as INGRP are one dimensional with the FOR_EACH.INGRP subscript. Line a1 states that the value of the key POINTER.ITEMREC is equal to the last element of IN.ITEM#.

The interim variables NEXT.ITEM#,POINTER.ITEMREC and END.INREC need not be described in the user supplied data statements. The dimensionality and name of parent nodes are implied, and higher level nodes are added to account for increased dimensionality. Implied dimensions are assumed to be virtual.

An assertion is referred to as a recursive assertion if the dependent variable is an element of an array and it depends, directly, or through a chain of assertions, on other elements of the same array. If the dependent variable element depends on elements in the same array with index values that are smaller than the value of the subscript used in the assertion, then the dependent variable elements can be evaluated progressively as the value of the subscript is incremented from 1 to the end of the range in steps of 1. This condition is checked, and if it is not satisfied then a warning message is issued and a Gauss-Seidel iterative procedure is generated to evaluate the dependent array variable elements.

Figure 3 contains a specification that illustrates the use of recursive assertions and referring to "historical" data, discussed previously. The variables of this specification form three structures. The input file IN, described in statements d1 to d3, contains the integer N which is to be tested for primality. The output file OUT constains an output record for printing the result which consists of a copy of N and a field DIV. DIV is a divisor if N is divisible (and hence non prime). If N is prime then DIV contains the alphabetic string 'PRIME'. The structure INT contains a table J in which integer products up to N are listed. J is a jagged

two dimensional array containing the history of products. J is an INTERIM FIELD with two virtual dimensions. The global subscripts of the array J are FOR_EACH.I and FOR_EACH.J.

The jagged matrix J is illustrated below for N=15.

	FOR_EACH.J						
FOR_EACH.I	1	2	3	4	5	6	7
1	4	6	8	10	12	14	16
2	9	12	15				

Note that only the value J(2,3) = 15 is of interest for finding DIV = FOR_EACH.J+1=3. The array is jagged, i.e. the range of the second dimension depends on the value of the first dimension subscript.

Since J is a two dimensional variable range array, END.J(also two virtual dimensions) and END.I (one virtual dimension) define the respective ranges. Since we are only interested in products not exceeding N we terminate the dimension associated with J when the value of N is exceeded. This is expressed in assertion a2. a3 is an example of a recursive assertion. It defines J. a4 and a5 define the variables DIV and N in the OUT file. The assertions, as stated in Figure 3, also

illustrate use of local subscripts. Following the above rules, subscripts can be omitted only in assertion a1.

3. REPRESENTATION OF A SPECIFICATION BY AN ARRAY GRAPH

As noted in the previous sections, much of the information needed for generating a program is implicit in the MODEL specification. It is therefore necessary to perform the analysis to make such information explicit. As a first step it is advisable to represent the specification in a convenient form, based on which implicit information can be derived and entered, checks be conducted and finally a schedule of program execution be derived. The conventional approach to this class of problems has been to use a form of a directed graph to represent dependencies and other relations involved in the computation. Similar to Petri Nets (Petri, 1962: Holt, 1960) and Data Flow Graphs (Dennis, 1973), our use of a directed graph is also mainly for the modeling of data dependencies. However, the straight forward approach of constructing a graph in which each computation of an array element is represented by a node is unacceptable. First, the number of elements in an array may not be available at the time of compilation, and secondly, the array [illegible] result in a huge unmanageable graph. [illegible] a new tool which we [illegible] aggregates are arrays.

represent potential processing steps associated with accessing and evaluating array variables. This means that each data structure and each equation (explicit and implicit) are represented by a node. This also means that each statement is represented by a node. When a file statement is designated (implicitly or explicitly) as both source and target data (where a file is updated) then separate nodes represent the source data and the target data. There are also nodes for the data parameter variables.

Each node is potentially compound, namely each represents the instances of the data structure or equation for all the array elements 1 to N. Information on dimensionality and range must therefore be associated with the nodes in the array graph. A node that corresponds to a data structure has associated with it subscripts that correspond to its dimensions. A node that represents an assertion (i.e. equation) has associated with it subscripts that corresponding to the union of subscripts of the variables appearing in the equation. Thus a compound m dimensional node A represents the elements from A(1,1,...1) to $A(N_1,N_2...N_m)$ where $N_1...N_m$ are the ranges of dimensions 1 to m respectively.

Similarly a directed edge may be compound in that it represents all the instances of dependencies among the

array elements of the nodes at the ends of the edge. These dependencies imply precedence relationships in the execution of the respective implied actions. There are several types of dependencies or precedences. For example, a Hierarchial (H) precedence refers to the need to access a source structure before its components can be accessed or, vice versa, the need to evaluate the components before a structure is stored away. Data dependency(D) precedence refers to the need to evaluate the independent variables of an equation before the dependent variable can be evaluated. Similarily, Data Parameters(P) precedence refers to the need to evaluate the data parameters of a structure (range, length, etc.) before evaluating the structure. Five such types of precedence relationships that are represented by directed edges in the array graph, are described more precisely in Table 1. These edges are determined based on the analysis of the information in statements associated with the respective end nodes. Since each edge may be compound it is necessary to associate with it information on dimensionality and ranges.

An array graph AG is then a pair (N,E) where N is a set of compound nodes and E is a set of compound edges. The array graph AG=(N,E) represents an underlying graph

1) Hierarchical (H): between a data node and its descendants in the data structure tree. For source data a node precedes its descendants; the opposite holds for target data.

2) Data Dependency (D): between an assertion node and its variable nodes. The independent variable nodes precede the assertion node which precedes the dependent variable node.

3) Data Parameters (P): between a data parameter variable node using keyword prefixes FOUND, END, ENDFILE, LENGTH, NEXT, POINTER and SIZE and the data node which is its subject (named in the suffix). For END, ENDFILE, SIZE, LENGTH, POINTER and SIZE keywords the data parameter node precedes the subject data node and vice versa for the FOUND and NEXT keywords.

4) Medium Order (M): between two sibling data nodes which are on an external file, reflecting the order of position of data on the file medium.

5) Virtual (V): Where the range of a dimension is denoted by an *, access to the I-1th element of a virtual dimension must precede access to the I th element. Thus, wherever there is a precedence relationship of types D or H between predecessor and successor nodes with a virtual dimension, there is also an edge in the reverse direction (labeled with the subscript expression-type 2: I-1) for each virtual subscript used in these nodes.

Table I: Edge Types

$UG=(N_u, E_u)$ which is a conventional directed graph where each instance of an element of an array is represented by a node and each instance of dependency is represented by an edge. The underlying graph UG is defined in terms of the array graph AG as follows. The nodes N_u of UG are :

$$N_u=\{A(I_1,I_2...I_n) \mid 1\leq I_1\leq N_{I1}, 1\leq I_2\leq N_{I2}, ...\text{where } A(I_1,I_2,...I_n)\in N\}$$

The edges in the underlying graph are between underlying graph nodes where the corresponding common subscripts have the same value. Let $A\leftarrow B$ be an edge E in the array graph AG, where A and B have common and different subscripts. Let the subscripts $I_1,...I_n$, be common to both nodes, while the subscripts $F_1, F_2,...F_k$ are exclusive to the A array and subscripts $G_1, G_2,...G_m$, are exclusive to the array B. The order of the subscripts of A and the subscripts of B is determined in the array graph AG. The underlying graph edges E_u which correspond to the array graph edge $A\leftarrow B$ are:

$E_u=\{A$(approp. ordered I and F subscripts)$\leftarrow B$(approp. ordered I and G subscripts)

$1\leq I_1\leq N_{I_1}, 1\leq I_2\leq N_{I_2}, \cdots$

$1\leq F_1\leq N_{F_1}, 1\leq F_2\leq N_{F_2}, \cdots$

$1\leq G_1\leq N_{G_2}, 1\leq G_2\leq N_{G_2}, \cdots$

Where $A\leftarrow B\in E\}$

Note that if there are no subscripts which are common to both A and B then the edges in the underlying graph are from every element of B to every element of A.

Two array graphs for the examples of Figures 2 and 3 are illustrated in Figures 4 and 5, respectively. Each array data node is represented by a dot labelled by the variable name and by its repetition specification, if it is a repeating structure. The graphs include nodes added by the system to reflect the dimensionality of data parameter variables. The assertions are represented by circles labelled by the assertion line number. Array edges are labelled by the edge type. However, in order not to clutter the diagrams excessively, V type edges are shown only in Figure 4, for only one of the virtual dimensions.

The data structures and assertions are stored by the MODEL processor in a simulated associative memory that facilitates search of a statement by variable names and keywords. A node directory is created based on the statements. The Hierarchical (H) and Medium (M) type edges are created first, followed by the Data Dependency (D).

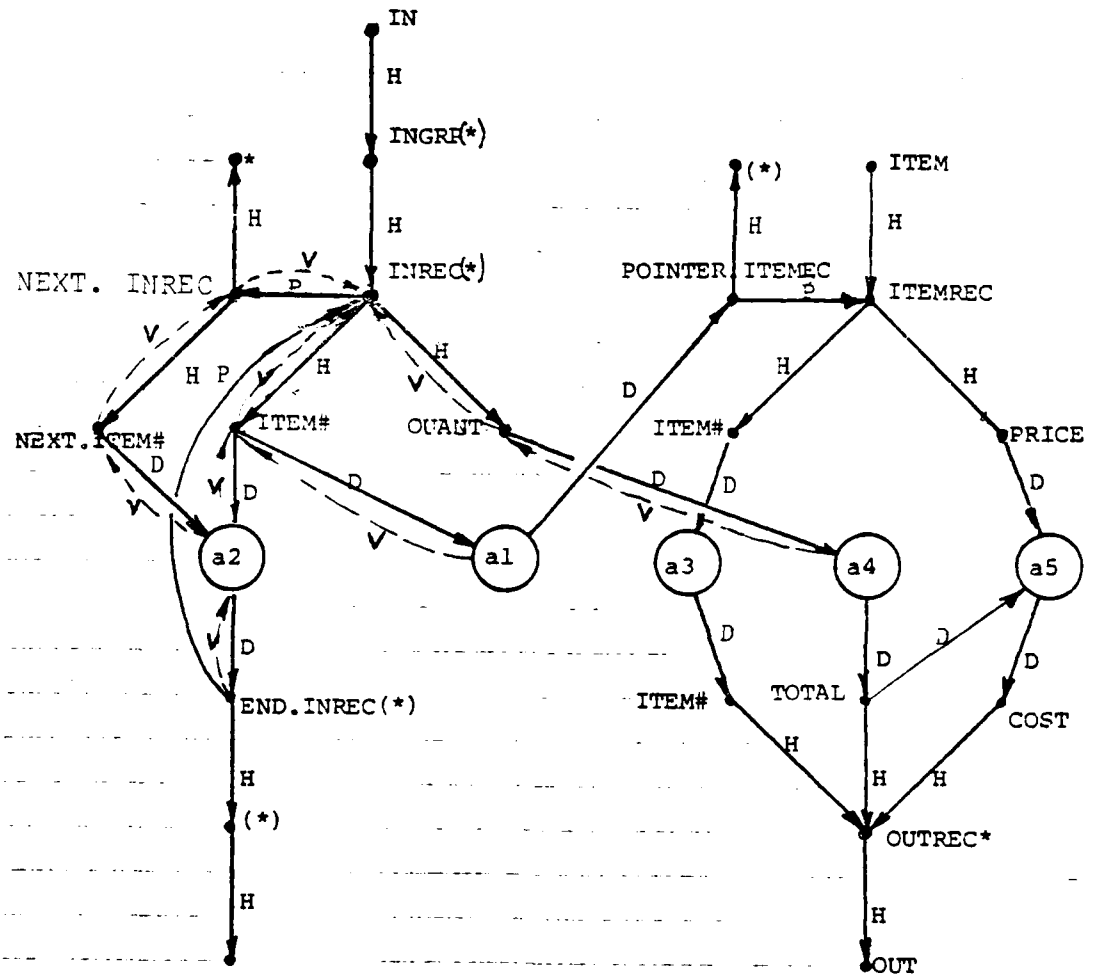


Figure 4 Array Graph of the Specification In Figure 2

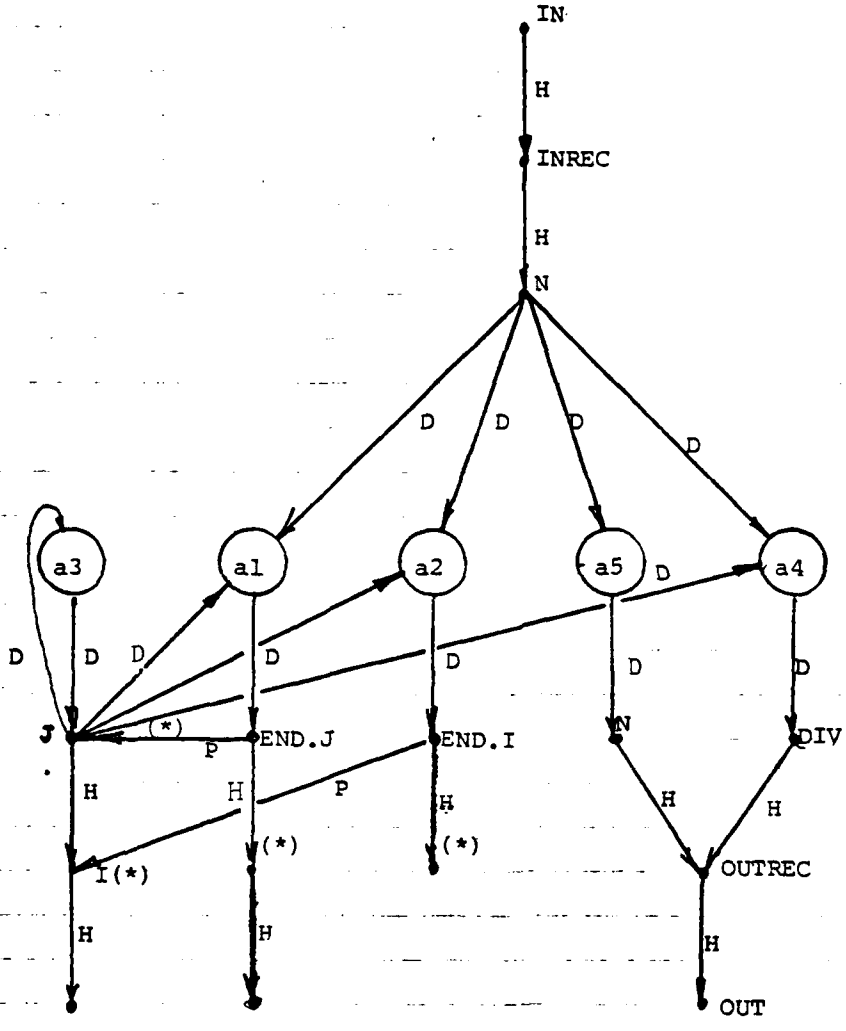


Figure 5 Array Graph For The Specification Of Figure 3

edges and Data Parameters(P) edges. Virtual type (V) edges are constructed during a later analysis phase. Later analysis may also indicate the need for additional nodes and edges. Data structures associated with nodes and edges are constructed at the time that the edges are created, but the values of some of the variables in these structures are determined later during the analysis phase. These data structures are presented in Tables 2-5 and will be referenced further in the discussion of the analysis of the array graph and the design of the corresponding program.

The array graph is represented by three data structures:

1) A node directory with a unique node number for each assertion and data (array) variable.

2) A node table - An entry for each node consists of the attributes associated with each node shown in Table 2 and attributes of the subscripts of the node shown in Table 3.

3) An edge table - consisting of the attributes associated with an edge shown in Table 4, and attributes of the subscripts of the edge shown in Table 5. Each edge structure constitutes an element in the two edge lists attributed respectively to the predecessor and successor nodes.

1. Node number and name
2. Node type: data|assertion.
3. If a node repeats: 3.1 Physical|virtual dimension.
   3.2 Range definition, if defined directly:
   |variable(SIZE/END arrays)
   |declared (subscript)
   |implicit (end of file marker)
   3.3 Node number of range specification if defined indirectly.
4. Apparent number of dimensions(D)
5. Local subscript list for subscripts associated with the node (see Table 3); ordered by dimension number (from left to right)
6. Sucessor Edges list
7. Predecessor Edges list

Table 2: Attributes Of A Node Structure

1. Position (dimension) number in node.
2. Is dimension reduced? Is there a reduction on that subscript (applicable only to assertion nodes)
3. Subscript form: FOR EACH.Y|SUB<n>|declared.
4. Node number of subscript declaration. (Each subscript declaration has its own node number)
5. Node number where range is defined directly.
6. Nesting level (if implemented by a nested loop).

Table 3: Attributes of an Entry in a Local Subscript List of A Node. (see Table 2, item 5)

1. Edge type H|D|P|M|V
2. Difference in number of dimensions between predecessor (p) and successor (s) nodes (δ).
3. Predecessor node number.
4. Successor node number
5. List of subscripts associated with the edge (see Table 5), ordered by position number in predecessor node.

Table 4: Attributes of an Edge s $\leftarrow$ p.

1. Local subscript position number in predecessor's node.
2. Local subscript position number in successor's node.
3. Subscript expression type: I|I-1|I-K|or other, (I-subscript, K > 1).

Table 5: Attributes of an Entry in a Subscript List Associated with an Edge

4. ERROR DETECTION AND CORRECTION OF A SPECIFICATION

4.1 Introduction

It is to be expected that a newly composed specification would contain ambiguities, imcompletenesses and inconsistencies, especially when the composer of the specification is not proficient in mathematics or programming. Since the system does not possess knowledge of the application, the automatic error detection and correction processes must depend only on the analysis of the inherent logic of the specification.

The program that is to be produced may be considered as transforming multi-dimensional data arrays into data arrays having the same or different numbers and ranges of dimensions. This requires compatibility of dimensionality and variables subscripting in assertions. If errors are found, we can do either of two things: correct the specification and warn the user, or, alternately, report an error and solicit a correction from the user. In either case the explanation of the problem discovered must be presented in terms of the nonprocedural specification and not in the procedural terms of the program that is being produced. We prefer to make corrections whenever reasonable and advise the user of such corrections as this facilitates explaining the problem that has been

detected. We realize that this approach is controversial in that designers of recent language processors frown on amending programs by default since this contradicts the notion of explicitness. In our case the warnings sent to the user emphasize and clarify the formalism that is being used.

Three general types of errors that may be detected and sometimes corrected are discussed here. Ambiguities arise from assigning the same name to several data structures. Recognition and correction of data name ambiguity is discussed in Section 4.2. Incompletenesses due to missing definitions for some of the variables are discussed in Section 4.3.[9] Inconsistencies arise when the assertions or definitions contradict themselves or one another due to incompatible dimensionality, ranges, subscripts, or due to circular definitions.

Inconsistencies can be identified in a three step process. The first step, dimension propagation, traces the array graph in order to determine consistent dimensionality of the nodes. Conflicts in dimensionality are either resolved or reported as errors. Dimension propagation is discussed in Section 4.4. Section 4.5 discusses the insertion of subscripts in assertions where they have been omitted. The last step, range propagation,

9 Shastry (1978) discusses extension of the completeness analysis to verifying that every element of an array is defined.

identifies the ranges of dimensions [illegible]
user has not provided specifications [illegible]
corresponding specified ranges [illegible]
This process also detects [illegible]
or missing range specifications. [illegible]
cribed in Section 4.6. [illegible]
in Section 5.

4.2 Ambiguity

The construction of edges proceeds from the [illegible]
data trees and traces the branches. The statements are stored
in a simulated associative memory from which statements may
be retrieved based on a boolean expression consisting of key-
words and data names. Thus, for instance, it is possible to
retrieve all the statements with the FILE keyword, and from
there to create H type edges which trace each data hierarchy
tree, and so on. When there are several data statements
with the same data name then there would be a corresponding
number of candidate edges for each precedence relationship
from or to the similarly named nodes. In assertions the
ambiguity must be removed by the user by prefixing each
ambiguous variable with the name of the appropriate ancestor.
The absence of such a prefix results in a corresponding error
message. In data statements the appropriate ancestor is implicitly
based on the order of the composition of the statements by the
user. When there is more than one statement with the same name,

the ancestor node statement that precedes it and is nearest to it in the order of composition of statements is selected as the assumed parent. Thus, for instance, in Figure 2 there are three data statements with an ITEM#. Following the order of statements it is possible to determine the parent of each. Otherwise it would have been necessary to use qualified names also in the data statements.

At the end of construction of H-type edges, any ambiguously named data which is not linked to a parent, is assumed redundant and is deleted.

4.3 Incompleteness

Incompleteness is the apparent omission of structures or assertions. If all the data and assertion arrays are defined then the array graph would be "complete" in the sense that an edge terminates and an edge originates at each node, except in the following special cases:

1) Source file statements and assertions that define variables by constants do not have edges that terminate at these nodes.
2) The nodes that represent target files do not have edges that originate at these nodes.
3) Some source field nodes may have no edges that originate at the nodes. In this case, the particular source data name is not used in an

assertion to define any other data and is only included for the complete specification of the data structure.

If the above completeness criteria are not satisfied, an appropriate data description statement or an assertion may be generated according to the following rules:

1) If the node under consideration represents a record, group or field of data, and the parent for that data name has been omitted by the user, then a parent data statement is generated. The array graph is also updated to include the parent-descendant relationship resulting from the generated statement. This allows a user to omit parent data statements especially in INTERIM data. Thus, for instance, in Figure 3 it is possible to omit the statements for INT and J (lines d8 and d9) and equivalent statements (using different names) would be generated by the processor.

2) If the node under consideration represents a target data field name, and, if no edge terminates at the node, then an assertion may be generated as follows: If there exists a source data, with the same name then we assume that

this source is to be copied into the identically named target variable. For example, if assertion a5 in Figure 3 was omitted, the assertion OUT.N = IN.N would be automatically added. All corrections are reported to the user in warning messages.

4.4 Dimension Propagation

Assertions generally transform multi-dimensional arrays, where the dimensionality of the arrays is indicated by the user through subscripting. However, as noted, some subscripts may be omitted by the user and are considered implicit. Furthermore the dimensionality of arrays implied in assertions must correspond to the dimensionality of those arrays specified in the respective data descriptions. If the declared number of dimensions of the data structure is too small then additional data statements are generated, otherwise an error message is sent.

The process of evaluating the number of dimensions of each node is performed in two steps. In the first step each edge is considered locally in order to evaluate 1) the difference (δ) between the numbers of dimensions of its predecessor (p) and successor (s) nodes (see Table 4, item 2), and 2) an apparent (initial) number of dimensions (D) of these nodes (see Table 2, item 4).

This step is performed during the construction of the edges of the array graph. The second step checks declared and apparent dimensionality of independent and dependent variables of each assertion and iteratively modifies the apparent number of dimensions until there is consistency of dimensionality throughout the array graph or an error is noted.

The evaluation of δ and D in the first step is as follows:

For type H edges:

for source data, if the successor (s) is a repeating data then $\delta=1$, else $\delta=0$;

for target data, if the predecessor(p) repeats then $\delta=-1$, else $\delta=0$.

D for data nodes is the number of dimensions as derived from analysis of the structure's data description. If the structure is not described, then D=0.

For type D edges (that originate or terminate at assertion nodes), the evaluation of δ and D is based entirely on the respective assertion as given by the user, and is independent of the dimensionality of its independent and dependent variables as specified in the respective data statements.

Consider a user provided assertion, a, with an independent variable X and a dependent variable Y.

a: $Y(I_k,....I_1)$=function($X(I_b.....I_a,J_m....J_1)$).

The I subscripts are distinct from the J subscripts. $\{I_a....I_b\}$ are subset of $\{I_1....I_k\}$. Then the apparent dimensionality of a, D(a) = k+m. For the edge a←X, δ=k-(number of subscript in $I_a....I_b$). For the edge Y←a, δ=-m. The evaluation of δ(a←X) and δ(Y←a) does not take into consideration the declared dimensionality of X and Y, respectively, but is derived only from the assertions.

To illustrate the above let

a: IF I = 2 THEN Y = X(I);

D(a)=1, then δ(a←X) = 0 and δ(Y←a) =-1

or if

a: Y(I,J)=SUM(X(K,J),K);

D(a)=3, then δ(a←X)=1, δ(Y←a)=-1.

For edges of type P δ=0, except in the case of P type edges SIZE.X←X δ=1, as the SIZE.X array is always of one dimension less than X.

The second step consists of repeated propagation of the dimensions throughout the array graph both forward and backward until either consistency is attained or an error is indicated. Propagation means that the number of dimensions of the node at one end of an edge is defined as equal to the number of dimensions of the node at the other end plus (minus, if backward propagated) δ. The direction of the propagation depends on the type of the

edge. The repeated propagations may either:

case a: converge—indicating consistency of dimensionality.

case b: diverge—with increasing number of dimensions of a node with each repeated propagation, until a bound is exceeded. This implies an error in dimensionality in some recursive assertion(s).

case c: the number of dimensions computed exceeds the number of dimensions of a declared output file. This implies an error either in data description or related assertions.

A simplified presentation of the algorithm is as follows. Let C(n) represent the current number of dimensions of node n. D(n) represents the initial (apparent) number of dimensions of node n. Let N denote the set of nodes and E the set of edges of the graph.

1. For all nodes $n \varepsilon N$ let $C(n) \leftarrow D(n)$
2. Repeat propagation of all edges until either:

case a: there is no change in C(n) for all $n \varepsilon N$,

or case b: any C(n), $n \varepsilon N$, exceeds a threshold (say 20) (error message),

or case c: for any data node which is not an interim variable or a field in a keyed file, $C(n) > D(n)$ (error message),

Repeat for each edge $e \varepsilon E$: $s \overset{e}{\leftarrow} p$

Propagate Forward:

(i) for H and D type edges,

(ii) for P edges terminating in ENDFILE, FOUND and NEXT prefixed data names,

(iii) for P edges emanating from POINTER prefixed data name

if $C(p)+\delta>C(s)$ then let $C(s)=C(p)+\delta$

Propagate Backward:

for P type edges emanating from END, LENGTH and SIZE prefixed data name

if $C(s)-\delta>C(p)$ then let $C(p)=C(s)-\delta$

3. Repeat for all $n \varepsilon N$

if n is an apex node of an interim structure, including keyword prefixed names, then generate statements that add C(n) dimensions to the structure.

4. Let $D(n) \leftarrow C(n)$

4.5 Filling Subscripts

At this point a consistent number of dimensions for each node (D, Table 2, item 4) has been determined. Also all the missing data statements have been generated. There remains the triple task of inserting:

1) entries for missing dimensions and subscripts in the local subscript list of respective nodes (see Table 3).

2) local subscripts in positions of missing subscripts in assertions.

3) entries for missing subscripts in the subscript lists of respective edges.

The addition of subscripts in node structures is as follows:

For data nodes the global form of subscripts FOR_EACH.X is used (Table 3, item 3). The subscripts are in the order of precedence in the respective data tree. For assertion nodes the local form of subscripts (S<n> or SUB<n>) is used (Table 3, item 3). The subscripts associated with an assertion node are ordered in accordance with the dimensions of the target variable followed by any reduced subscripts.

Local subscripts are inserted in the assertions. Subscripts are added from right to left (S1, S2, etc.), until all the dimension positions are filled. For example assertion a4 in Figure 2:

TOTAL = SUM(QUANT(FOR.EACH.INREC), FOR_EACH.INREC)

would be modified to:

TOTAL (S1)=SUM(QUANT(S1, FOR_EACH.INREC), FOR_EACH.INREC).

As TOTAL is one dimensional and QUANT is two dimensional, S1 has been added to both on the left side.

Finally, edge subscript structures (Table 5) are added to the edges emanating from the nodes where subscripts were added.

4.6 Range Propagation

A range of a dimension of a node (data or assertion) may be specified directly in statements associated with the node or indirectly through range propagation. There are four ways to define a range of a dimension directly (see item 3.2 in Table 2)

1) Fixed: through specifying an integer number of repetitions of the respective data statement.
2) Variable: Through defining an array with the SIZE or END prefix names and the node name as suffix.
3) Declared: through a data statement of a subscript name, including the number of repetitions.
4) Implicit: through end-of-file marker of a source sequential file.

It would be cumbersome for the user to define the range of each dimension of each node. Therefore, in the absence of a range specification for a dimension of a variable, the assertions where the variable is used are analyzed for implication of the range. For example, the assertion $X(I_m...I_1) = Y(I_m...I_1)$ may imply that the ranges of the dimensions in X and Y referred to by the same subscript name are the same. This is referred to as range propagation. The range in this case is defined indirectly through propagation of the range from another data node. If a range is specified indirectly,

then the node number where the respective range is defined directly is given in the node structure (as shown in Table 2, item 3.3).

The function of the range propagation process is to determine the range sets, namely the sets of nodes and respective positions that have a common range definition.

Consider an edge e: s←p. The correspondence of respective dimensions in nodes p and s is given in the subscript entries associated with the edge e (see Table 5, item 1 and 2). For subscript expression of types 1,2 and 3 (I, I-1 or I-K, see Table 5, item 5) and in the absence of contradictory range specifications, the indicated corresponding subscripts in p and s are assumed to have the same range and be members of a corresponding range set. By repeated propagations, a range set is determined, consisting of node-number and position-number pairs which have only one common range specification. Note that the range is not propagated where a subscript expression is of type 4 (i.e. constant or any other form differing from types 1,2 and 3). If there are more than one same range specification for a range set then the specifications are redundant and all but one could be deleted or disregarded, and a warning message issued. It there is no range specification then an error message is issued.

The examples in Figures 4 and 5 amply illustrate range propagation. For example, in Figure 4 the second dimension of INREC is specified directly by an assertion defining END.INREC. This range is propagated through H and D type edges to the second dimension of ITEM#,QUANT, a1, a2, a4, NEXT.INREC, NEXT.ITEM# and END.INREC. Requiring the user to provide range specifications for all these nodes would have been unacceptably tedious.

The algorithm for performing the range propagation follows:

1. Determine the nodes with direct range specifications: Place all node-dimensions where the range specification is direct on a list L.

2. Propagate range of dimensions: For each node in L, the specified range is propagated forward through emanating series of edges and backward through the terminating series of edges until the appropriate dimension is found to be reduced or a conflicting directly specified range is encountered. The node number and dimension number of each traversed node is entered into a range set corresponding to the specified range in the node in L. In tracing the edges, if a traversed node is a data node where the range-propagated dimension is declared as repeating (in the corresponding data statement) but the range

is defined indirectly, then the node number of the starting node in L is entered in item 3.3, Table 2.

3. Issue error message: Determine all data nodes where the rightmost dimension is defined as repeating (see item 3.1, Table 2) but the range is undefined (item 3.3, Table 2) and report them as missing specifications of number of repetitions.

The V type edges are constructed while the virtual dimensions ranges are propagated. There would be a V type edge in the reverse direction for each virtual subscript associated with H or D type edges, and for P type edges emanating from a POINTER prefixed data names. The subscript expression of type 2 (I-1) is associated with the virtual subscript of a V type edge (see Table 5, item 3) to denote precedence of the previous element.

5. FLOWCHART DESIGN

At this point in the compilation process, the specification is assumed to be complete, nonambiguous and consistent. The next step is then to produce a flowchart of the actions to be taken by the program. The flowchart is an intermediate, object language independent, skeletal representation of the program. Recall that the nodes of the array graph represent accessing and computing actions and the edges indicate necessary precedence requirements between actions represented by nodes. The flowchart is essentially a linear arrangement of nodes according to the partial order imposed by the edges. The final code-generation phase of the processor (not described in this paper) essentially translates individual entries in the flowchart into blocks of code in the object language (presently PL/1 or Cobol).

There are two special interdependent problems that must be coped with in generating a flowchart. First, the array graph may contain cycles which prevent ordering the nodes in accordance with the edges. A maximally strongly connected component (MSCC) results from cycles in the array graph. Such cycles are illustrated in Figures 4 and 5. The V type edges create an MSCC consisting of all the nodes that have a virtual dimension. P type edges

emanating from the END.INREC and END.J nodes and the recursive assertion a3 also create cycles in the array graphs. A set of simultanous equations also forms a MSCC.

Secondly, each node represents an array of data or equations and it is necessary to assure that all the elements are individually accessed and evaluated. Consider the simple example of a single node consisting of assertion a:

$$A(I_1.....I_n)=f(B(I_a...I_b,J_1...J_m))$$

The I and J subscripts are distinct. $I_a....I_b$ is a subset of $I_1...I_n$. Assume that Cond.I_1...Cond.I_m recognize the last elements in the ranges of $I_1...J_m$. To evaluate all the elements of assertion a it may be bracketed by iteration statements for all it's subscripts. The elements will then be evaluated while progressively varying the indices in each dimension from 1 to the last element, as follows:

```
do I1 while cond. I1;

    .
    .
    .
      do In while cond.In;
         do J1 while cond.J1;

            .
            .
            .
              do Jm while cond.Jm;
                    a;
              end Jm;

                 .
                 .
                 .
            end J1;
         end In;

            .
            .
            .
      end I1;
```

Much of this section is concerned with analysis related to the above two problems.

The general approach to scheduling consists of creating a component graph which consists of all the MSCCs in the array graph and the edges connecting the MSCCs. The component graph is therefore an acyclic directed graph. It is then topologically sorted, resulting in a linear arrangement of the components which can be regarded as a gross level representation of the flowchart. The subscripts for each component are determined and appropriate iterations for these subscripts bracket the respective components. Finally each component is analyzed in greater depth to determine a suitable method for its evaluation.

We essentially employ two methods for scheduling the evaluation of a MSCC. In the first method an attempt is made to decompose the MSCC by deleting appropriate edges. Consider the simple example of a two node MSCC consisting of a one dimensional array X and the assertion a: X(I) = X(I-1)+1. I is a subcript common to both nodes and N is the range of I. Therefore the schedule would be:

```
do I from 1 to N
     MSCC consisting of nodes a and X
end I
```

The edge a←X has associated with it a subscript I of type 2 (I-1). It indicates that evaluation of the I-1 th element of X must precede the evaluation of the I th element. But this is already assured by the order of iterations for I from 1 to N. Therefore this edge may be deleted, which may cause decomposition of the MSCC and allow for its scheduling. More generally, to decompose a multi-node MSCC it is necessary to:

1) Find a dimension and position in each node of the MSCC which all have a common range that can be given a corresponding common subscript name.to use in an iteration statement that brackets the entire block of nodes that constitutes the MSCC.
2) Find edges that represent dependencies on lower index elements of the selected subscript; these edges are deleted and may cause decomposition of the component.

For complex MSCCs the decomposition and scheduling may be performed recursively until all the cycles are opened.

If no suitable subscript is found or if no edge can be deleted, then the user is advised of this and an iterative solution method is employed, typically the Gauss Seidel method. For instance, consider an MSCC consisting of the scalars X and Y and the two assertions X= aY+b; Y=cX+d. No decomposition of the MSCC is feasible in this case The processor therefore incorporates in the

program an iterative method to solve these equations. The user then must check the convergence of the solution. The part of the MODEL language for this task has been omitted in this paper.

At the end of the scheduling process, an optimization process further attempts to consolidate adjacent blocks of nodes which are iterated over the same range. This increases the scope of the iteration and improves the efficiency of the resulting program.

The SCHEDULING procedure consists of two procedures, SCHEDULE-GRAPH and SCHEDULE-COMPONENT, which are mutually recursive.

SCHEDULE-GRAPH finds the MSCCs and topologically sorts the component graph. It is given two arguments: 1) the graph to be scheduled (g), and 2) the level of the recursive call (ℓ) corresponding also to the level of iteration loop nesting. It returns a schedule of the nodes of the graph, $(s_1....s_n)$.

SCHEDULE-COMPONENT analyses and decomposes an MSCC. It is given two arguments: 1) a MSCC (g_i) to be decomposed and 2) the level of recursion (ℓ). It returns a block of nodes bracketed by the iteration parameters and the level of nesting of the iteration.

SCHEDULING is initiated by calling SCHEDULE-GRAPH with the arguments: g, being the entire array graph,

and ℓ=0.

The algorithm of SCHEDULE_GRAPH is as follows:

1. Find all the MSCCs. This is done using the depth first search algorithm (Tarjan, 1972).

2. Sort topologically the MSCCs into a linear order $g_1 \ldots\ldots g_m$.

3. Remove edges in g between g_i and g_j, $\forall i, j: i \neq j$. This deletes the edges connecting MSCCs. Such edges are not needed further.

4) Repeat for each g_i, i=1 to m

s_i = SCHED_COMPONENT(g_i, ℓ). s_i is the ith component (single or multi node) in the flowchart. This calls the SCHED_COMPONENT process for each component.

5) Return the flowchart $s_1 \ldots s_m$. This constitutes the final result.

The algorithm of SCHED_COMPONENT is as follows.

1. Determine candidates for subscripts for bracketing the component: the smallest set of <u>available</u> dimensions in g_i is determined. These are the dimensions of a node in g_i which has the smallest number of dimensions which also have not been selected previously (for smaller values of ℓ). Let the selected node be M. Let m= number of available subscripts in M.

2) Return a single node as a schedule element: if m=o and the number of nodes in g_i=1 then return g_i as a

schedule element. $s_i=g_i$:

3) Report a non-decomposable MSCC: if m=o (no available subscripts) and the number of nodes in $g_i>1$ (i.e. a multi node MSCC) then this is a non-decomposable cycle in the graph. There are several possible causes of a non-decomposable MSCC as follows:

1) if the MSCC contains V type edges then this indicates that it is not feasible to implement the user specification of the corresponding virtual dimension. The respective dimension must then be changed to a physical dimension.

2) if the MSCC contains at least one edge of types H,P or M, then there is a mathematical inconsistency caused by circular logic or incompatibility in dimensionality or subscripting. This is considered a user error.

3) if all the edges in the MSCC are of D type, and the number of assertion nodes in the MSCC equals or exceeds the number of data nodes then the problem may be due to simultaneous equations or because the dependencies of elements of the arrays are not in descending order of element index values. This then suggests that an iterative solution, such as the Gauss Seidel method, is called for. g_i is altered to form a graph corresponding to such an iterative solution procedure and step 8 is executed next.

The description of generating an iterative solution procedure is beyond the scope of this paper (Gana, 1978). Messages are issued identifying the nodes in the MSCC and the indicated problem.

4) Select a common range (and subscript) for an iteration to bracket the nodes in the MSCC: Starting with M, repeatedly propagate the range for each of the available dimensions (similar to the range propagation in Section 4) until a chain of same range dimensions (a range set) is found where the range is propagated to only one dimension of every node in g_i. Available dimensions that do not satisfy this condition are marked as not available.

5) Name the selected subscript: The highest order subscript of M which satisfies the above criteria is selected and a subscript name is associated with it. The selected subscript is noted as unavailable. A selected subscript range must not depend on yet unselected subscripts. Also virtual subscripts of sequential files must be selected in the order of dimension positions. An error message is issued if these conditions are not satisfied indicating an inconsistency in subscrip-

ting.

6) Remove edges: All edges of expression types 2 and 3 (I-1, I-K) of the selected subscript are deleted from the MSCC g_i.

7) Enter the value of ℓ and the selected subscript name in subscript entries of the nodes in g_i (see Table 3, item 6).

8) s_i = SCHEDULE_GRAPH(g_i, ℓ+1). This returns the decomposed MSCC for a recursion of scheduling.

9) Bracket the schedule returned by SCHEDULE_GRAPH: The returned schedule consists of, one or several elements. A block is formed by bracketing these elements within an iteration for the selected subscript, if any.

10) Return the bracketed block as a schedule element.

After obtaining a schedule for the array graph, the further OPTIMIZATION procedure endeavors progressively to enlarge the scope of iterations and thereby attain a more efficient program. The algorithm of OPTIMIZATION consists of progressively evaluating adjacent blocks in the schedule as candidates for consolidation. The condition for consolidating adjacent blocks A and B are:

1) The ranges of the iterations that bracket the blocks A and B are the same.

2) The dimensional positions of the same range dimensions in the independent variables (rhs) of

the nodes in A are the same as the dependent variables (lhs) of the nodes in B. This condition checks for instance that block B does not depend on a transposed array which is defined in A, in which case blocks A and B cannot be within the scope of a single iteration for the respective subscripts.

The above algorithms are illustrated in the flowcharts in Figures 6 and 7 for the examples in Figures 4 and 5 respectively. The initial topological sorting of the MSCCs in the graph of Figure 4 by SCHEDULE_GRAPH results in the ordered list of 8 components. These components are listed on the following lines of Figure 6: 1,2,3, a MSCC for i=1: lines 5-25, 27, 28, 29, 30. SCHEDULE_COMPONENT is then called for each of these components. For the first three components, and later for the last four, m=0 and therefore they are reported as schedule elements. The next component (shown in lines 5-25) is a MSCC including all the V type edges for the virtual subscript FOR_EACH.INGRP. The global subscript FOR_EACH.INGRP is selected as an iteration parameter. The MSCC is bracketed by iteration statements for FOR_EACH.INGRP and all the edges with subscript expressions of FOR_EACH.INGRP of types 2 and 3 are deleted. The V type edges for FOR_EACH.INGRP have a subscript expression of type 2 and are therefore deleted. SCHEDULE_COMPONENT then calls SCHEDULE_GRAPH recursively to schedule the sub-

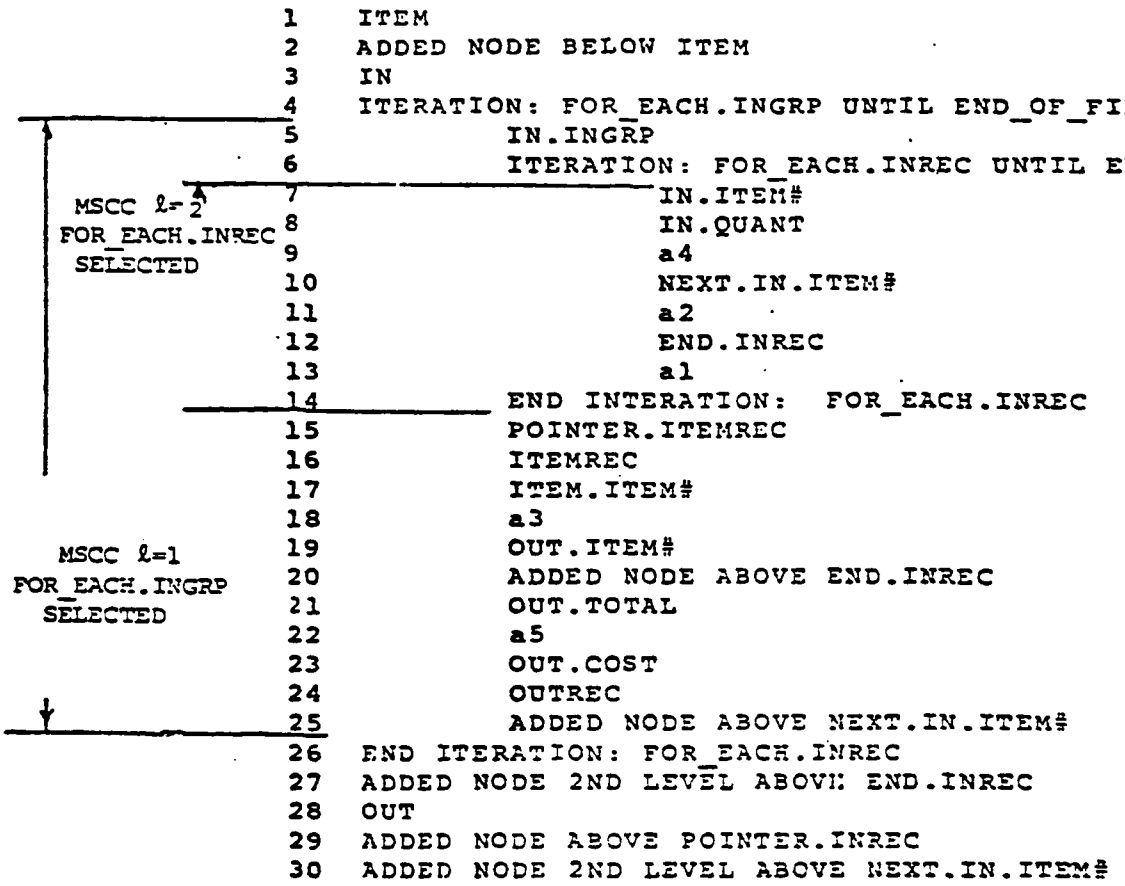


Figure 6. Flowchart Generated For The Example In Figure 2

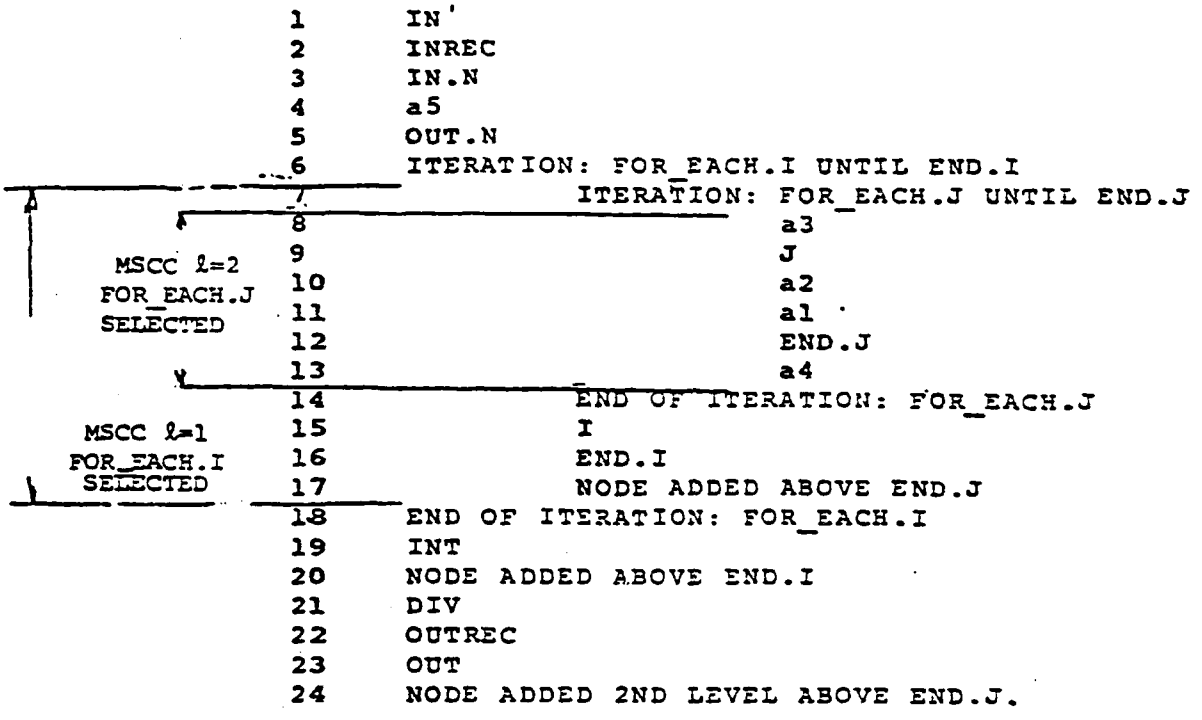


Figure 7 Flowchart Generated For The Example In Figure 3

graph of the MSCC with the deleted edges. SCHEDULE_GRAPH topologically orders the components of the new subgraph. This results in 5 components shown on lines: 5, a MSCC for ℓ=2 lines 8-13,15,16 and 17. SCHEDULE_GRAPH further calls SCHEDULE_COMPONENT for each of these components, now with ℓ=2. For each iteration nesting level there are further recursive calls on SCHEDULE_GRAPH and SCHEDULE_COMPONENT until all respective MSCCs are decomposed into single node schedule elements.

A similar process would produce the flowchart of Figure 7 based on the array graph of Figure 5.

6. CONCLUSION

As stated in the introduction, the goal of the MODEL project has been the development of a nonprocedural language system with the characteristics of 1) automatic handling of all input/output activities, 2) global checking of completeness and consistency and 3) compiling. As shown, these three characteristics are mutually supportive in achieving a practical and useful system.

This article is in a sense a progress report, although the development has been underway for the past 5 years. The presently described algorithms represent an approach to a system that is tolerant of many types of users' ambiguities, incompletenesses and inconsistencies, and, at the same time, explicitly reports the semantics of the interpretation of the program specification to the user.

The two features, speedier program development and global logical checking, would make possible some new applications of computers, especially where a large number of programs are required quickly and inexpensively or where extensive debugging based on running the programs is normally needed. We had some experience with the former situation in a project where many business oriented programs had to be developed and given to key companies so that they could generate formatted reports for the Internal Revenue Service based on their own diverse and private data bases (Prywes, 1977). We are currently investigating a significantly different application where the system would be used in online economic

forecasting (Gana, 1978). The concept in this case is that global checking and correction of the specification would reduce the amount of debugging presently being experienced in economic modelling and forecasting and that very large models (up to 20,000 equations) could be executed much more efficiently than with the interpretive economic modelling systems. This type of application requires extensions in three main areas, on which research is proceeding. These are: 1) numerical solution of simultaneous equations, 2) extending the language to allow matrix algebra equations and, generally, operations on high level data structures and 3) modularization of a MODEL specification so that submodule programs may be independently generated and executed in distributed computers.

Another area of research concerns optimization of memory in the produced programs and, in particular, determining automatically which dimensions may be considered virtual, in the sense of this article.

ACKNOWLEDGEMENT

Mr. Kang-Sen Lu has participated in the development and testing of the version of the MODEL system which incorporates the algorithms described in this article.

7. REFERENCES:

1. Ashcroft, E.A. and Wadge, W.W., "Lucid, A Nonprocedural Language With Iteration," Communications of the ACM, Vol. 20, No. 7, July 1977, pp. 519-526.

2. Dennis, J.B., "First Version Of A Data Flow Procedure Language," MAC Technical Memorandum 61, M.I.T., Project MAC, Cambridge, Mass., May 1975.

3. Deo, N. and Mateti, P., "On Algorithms For Enumerating All Circuits of A Graph," SIAM Journal of Computing, Vol. 5, No. 1, March 1976.

4. Gana, J.L., "Use and Extension of An Automatic Program Generator For Model Building In Social and Engineering Sciences," Ph.D. Dissertation In Computer and Information Science, University of Pennsylvania, Philadelphia, Pa., 19104, 1978.

5. Holt, A.W., "Introduction To Occurance Systems," in Associative Information Techniques, Jacks E.L., ed., Americal Ellservier, New York, 1971, pp. 175-202.

6. Johnson, D.B., "Finding All The Elementary Circuits Of A Directed Graph," SIAM Journal of Computing, Vol. 4, No. 1, 1975, pp. 77-84.

7. Langefors, B., "Information System Design Computaticns Using Generalized Matrix Algebra",BIT 5(2), 1965.

8. Leavenworth, B.M and Sammet, J.E., "Overview of Non-Procedural Languages," Proceedings Of The Symposium On Very High Level Languages, SIGPLAN notices, ACM, April 1974.

9. "MODEL II - Automatic Program Generator, User Manual," Revision of Version 3, Contract TIR-77-41, Office of Planning and Research, Internal Revenue Service, Washington, D.C., January 1978.

10. Nunamaker, J. F., "On The Design and Optimization of Information Processing Systems," Ph.D. Thesis in Operations Research, Case Western Reserve University, Cleveland, Ohio, 1969.

11. Petri, C.A., "Kommunikation Mit Automaten," Schriften des Rheinisch - Westfalischen Institutes For Instrumentelle Matematik an der Universitat Boon Hft. 2, Bonn 1962.

12. Pnueli, A., K. Lu and N. Prywes, "MODEL Program Generator: System and Programming Documentation", Technical Report, Moore School of Electrical Engineering, University of Pennsylvania, 1980.

REFERENCES (continued)

13. Prywes, N., A. Pnueli and S. Shastry, "Use of a Non-procedural Specification Language and Associated Program Generator In Software Development," ACM Trans. On Programming Languages and Systems, Vol. 1, No. 2, October 1979, pp. 196-217.

14. Prywes, N., "Reference Manual For Use of MODEL In The IRS Tape Program". Report to the Planning and Research Division, Internal Revenue Service, Washington D.C., November 1977.

15. Tesler, L.G., and Enea, H.J., "A Language Design For Concurrent Processes", Proc. Spring Joint Computer Conf. AFIPS Press, 1968, pp. 403-408.

16. Shastry, S., "Verification and Correction of Non-Procedural Specification In Automatic Generation of Programs", Ph.D. Dissertation In Computer and Information Science, University of Pennsylvania, Philadelphia, Pa., 19104, 1978.

17. Tarjan, R.E., "Depth First Search and Linear Graph Algorithm," SIAM Journal of Computing, Vol. 1, No. 2, 1972, pp. 140-160.